VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO HODNOCENÍ ÚHLEDNOSTI ZDROJOVÝCH SOUBORŮ

THE TOOL FOR ASSESSING THE NEATNESS OF SOURCE CODE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **DAVID JAHODA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. PETR VEIGEND**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Jahoda David**
Program: Informační technologie
Název: **Nástroj pro hodnocení úhlednosti zdrojových souborů**
The Tool for Assessing the Neatness of Source Code
Kategorie: Analýza a testování softwaru

Zadání:

1. Seznamte se s existujícími možnostmi hodnocení úhlednosti a dodržování programovacích konvencí v různých programovacích jazycích. Nastudujte jednoduché metody statické analýzy zdrojových kódů v jazyce C.
2. Navrhněte nástroj, který bude hodnotit dodržování zvolených programovacích konvencí a úhlednosti zdrojových kódů v jazyce C. Programovací konvence by mělo být možné jednoduchým způsobem konfigurovat. Nástroj by měl umožňovat automatické hodnocení kvality zdrojových kódů týkající se úhlednosti a programovacích konvencí.
3. Navržený nástroj implementujte. Klaďte důraz na jednoduché použití. Nástroj by měl kvalitu hodnotit na základě použití základních technik statické analýzy.
4. Funkcionalitu nástroje podpořte umělými testovacími případy. Demonstrujte přínos nástroje na sadě studentských projektů.

Literatura:

- Clang C Language Family Frontend for LLVM (https://clang.llvm.org/)
- https://cs50.readthedocs.io/style50/

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Veigend Petr, Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce se zabývá tvorbou nástroje, jenž by umožňoval kontrolu a hodnocení úhlednosti zdrojových kódů v jazyce C. Primární cílovou skupinou jsou studenti předmětu Základy programování. Způsob realizace uvažuje využití nástroje Clang-Tidy, rozšířeného o vlastní sadu kontrol a programu vyhodnocující výsledky kontrol na základě vstupní konfigurace. Vytvořený program je schopen bodově hodnotit zdrojové kódy za pomoci 16 kontrol dle konfigurace. Tyto kontroly odhalují různé začátečnické chyby. Z testování studentských projektů vyplynulo, že nejčastější chybou je využívání tzv. magických čísel. Program je možno s vhodným poučením studentů nasadit v předmětu Základy programování (IZP).

Abstract

This work deals with creation of tool that would allow the checking and evaluation of neatness of source codes in the C language. The primary user group are students of Introduction to Programming Systems (IZP). The implementation considers the use of Clang-Tidy tool (extended with custom set of checks) and program that evaluates results of checks based on the input configuration. The created program is capable of scoring source code using 16 checks according to the configuration. These checks detect various beginners errors. Testing of the student projects revealed that the most common error is the use of so-called magic numbers. The program can be deployed in the Introduction to Programming Systems (IZP) course with appropriate student instruction.

Klíčová slova

Úhlednost, statická analýza, zdrojové kódy, programovací konvence, jazyk C, clang-tidy

Keywords

Neatness, static analysis, source codes, coding conventions, C language, clang-tidy

Citace

JAHODA, David. *Nástroj pro hodnocení úhlednosti zdrojových souborů*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Petr Veigend

Nástroj pro hodnocení úhlednosti zdrojových souborů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Veigenda. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Jahoda
12. května 2021

Poděkování

Tímto bych rád poděkoval vedoucímu mé práce, Ing. Petru Veigendovi za odborné vedení, pomoc a užitečné rady při řešení této práce.

Obsah

Kapitola 1

Úvod

Tato práce si klade za cíl vytvořit nástroj pro hodnocení úhlednosti zdrojových kódů v jazyce C. Jeho tvorba je motivována snahou vštípit začínajícím programátorům již od počátku návyky pro psaní úhledného kódu. Výsledná aplikace pak bude využita v rámci předmětu Základy programování (IZP), který je vyučován v prvním ročníku na FIT VUT.

Úhledným kódem je myšlen takový kód, který je přehledný, srozumitelný a snadno udržovatelný. K tomu by měl dopomoci tento nástroj tím, že ve vstupním zdrojovém kódu vyhledává typické začátečnické chyby, jenž porušují úhlednost, informuje o nich a kód jako takový hodnotí.

Tato práce vychází ze sdíleného zadání, jenž bylo rozděleno na dvě samostatné práce. Tato se zabývá takovou skupinou chyb, jenž lze odhalovat pomocí metod statické analýzy. Tyto chyby, ačkoliv jsou syntakticky korektními konstrukcemi, tak svou funkcionalitou zhoršují výslednou čitelnost a přehlednost kódu (např. příliš mnoho větvení, nepodmíněné skoky, zřetězené volání funkcí, atd.). Druhá práce, kterou tvoří kolega Ondřej Kinšt, se zabývá chybami syntaktického charakteru (jednotnost pojmenovávání proměnných, odsazování, maximální délka řádku, apod.).

Jako první je rozebrán teoretický základ obsahující uvedení do problematiky úhlednosti, rozbor aktuálně dostupných řešení na poli analýzy a hodnocení kódu, následované kapitolou věnovanou statické programové analýze a nástroji Clang-Tidy. Poté následuje popis navrhovaného řešení, tj. rozšíření Clang-Tidy o nové kontroly a vytvoření programu, jenž zpracovává výsledky Clang-Tidy a hodnotí zkoumané kódy. Na něj pak navazuje kapitola o implementaci popisující realizaci výše zmíněného návrhu. A na závěr je implementované řešení testováno nad sadou umělých i reálných příkladů.

Kapitola 2

Teorie

Tato kapitola pojednává o teoretickém základu, popisu nástrojů a jejich principů na kterých je založena tato práce.

2.1 Problematika úhlednosti zdrojových kódů

Tato část kapitoly vychází ze sborníku Using a coding standard to improve program quality [4].

Úhlednost nebo též čitelnost kódu je jednou ze základních podmínek pro udržovatelný kód. Úhledný kód též podstatně usnadňuje práci vícero programátorů zároveň na stejném projektu. Úhledný kód je také nezbytností pro možnou budoucí účast nových programátorů, jenž nebyli přítomní v době vývoje, na následné údržbě programu.

Nicméně vzhled psaného kódu je do značné míry závislý na konkrétním programátorovi. Každý programátor je totiž různě zkušenný a má i jiné návyky psaní kódu. Tyto návyky mohou již sami o sobě vést k nečitelnému kódu anebo to může způsobit střet různých programovacích návyků v jednom softwaru.

Jedním z hlavních nástrojů pro dosažení úhledného kódu je zavedení a dodržování kódovacích konvencí (např. Google C++ Style Guide[1]), stylů kódování (např. PEP 8[2] pro jazyk Python) anebo přímo standardů kódování (např. MIRSA[3]). Tyto konvence ustanovují mnoho aspektů výsledného kódu jako např. formátování (způsob odsazování, pojmenovávání proměnných/funkcí, ...), funkční aspekty (např. zakázané konstrukce jako nepodmíněné skoky, rekurze, atd.) a další. Tyto konvence se pak vymáhají posuzováním kódů (angl. code review) a to zpravidla automatizovaně.

Tyto všechny kroky pak mají za výsledek snadno udržovatelný kód, který je tím pádem i úhledný.

2.1.1 Požadavky na nástroj pro kontrolu úhlednosti

Nástroj by měl umožňovat:

- Analyzovat zdrojové soubory psané v jazyce C.
- Zpravovat uživatele o výsledku (výstup na standardní výstup, výstup do souboru, ...) a to vhodným způsobem (výpis názvu chyby, místo v kódu, ...).

[1] https://google.github.io/styleguide/cppguide.html
[2] https://www.python.org/dev/peps/pep-0008/
[3] https://www.misra.org.uk/

- Hodnotit vstupní zdrojový kód a to buď slovním či bodovým ohodnocením.
- Konfigurovat jeho nastavení (které kontroly spouštět, jejich váhu v hodnocení, parametrizaci, ...).
- Snadné nasazení/instalace (např. vhodný instalační skript či instalátor, ...).
- Jednoduché používání (např. minimální počet parametrů u terminálové aplikace, GUI anebo webová aplikace, ...).

Na základě těchto požadavků následně probíhalo hledání vhodného řešení a byly také určující ve fázi návrhu a implementace řešení.

2.1.2 Dostupná řešení

Při zkoumání dostupných řešení pro analýzu kódu a dodržování určitých programových konvencí jsem dospěl k jejich rozřazení do tří pomyslných skupin. A to jmenovitě na integrované, proprietární a úzce specializované. Následuje jejich rozbor a zhodnocení jejich vhodnosti pro účely práce.

Integrované řešení

Jedná se o řešení přímo integrované ve vývojovém prostředí (IDE - Integrated Development Environment). Typickými zástupci jsou řešení firem Microsoft (Visual Studio) a JetBrains (CLion, PyCharm, ...). Tato řešení se zpravidla zabývají spíše syntaktickou stránkou kódu. Tj. formátování kódu, syntaktické chyby obecně, či navrhováním drobných optimalizací (např. vhodnější pojmenování proměnné, úprava parametrů, nedosažitelný kód, chybějící přetypování apod.) [7][5].

Pro účely práce jsou tato řešení nevhodná z důvodu nemožnosti rozšiřování o nové vlastní kontroly. A dále také to, že pro použití kontrol je třeba instalovat celé vývojové prostředí, které může být pro začátečníka zbytečně komplikované.

Proprietární řešení

Další skupinou jsou různá proprietární řešení, např. SonarQube. Nutno podotknout, že verze SonarQube pro jazyk C potažmo C++ je zpoplatněna [11], což je trend vyskytující se i u dalších řešení tohoto typu jako například Coverity[4] od firmy Synopsys nebo Klocwork[5] od firmy Perforce.

Kvůli zpoplatnění a u některých i uzavřené povaze, mohou být tato řešení obtížně začlenitelná do vlastních celků a obecně i problematická co se týče jejich rozšiřitelnosti. Tyto nástroje proto nejsou z mého pohledu vhodným řešením.

Úzce specializované řešení

Další rozsáhlou skupinou řešení jsou nástroje, které se úzce zaměřují na určitou specifickou oblast chyb. Jedná se například o hledání nebezpečného kódu (přetečení zásobníku, chyby s nulovým ukazatelem apod.) či třeba hledání konkrétních bezpečnostních chyb. Zástupci této skupiny jsou třeba Facebook Infer anebo Frama-C.

[4] https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html
[5] https://www.perforce.com/products/klocwork

Nicméně je možné do této skupiny zařadit i řešení Pylint či CodeChecker, jenž už se požadavkům této práce blíží podstatně více.

Pylint

Pylint[6] je nástroj typu linter, pro kontrolu dodržování pythonovského standardu PEP 8[7]. Pylint může také přímo vyhledávat chyby, případně nabízet možné refaktorizace kódu. Taktéž umožňuje konfiguraci spouštěných kontrol, tvořit statistiky ohledně chyb a varování, či též bodově hodnotit zkoumaný kód [9].

```
robertk01 Desktop$ pylint simplecaesar.py
************* Module simplecaesar
simplecaesar.py:3:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
simplecaesar.py:1:0: C0114: Missing module docstring (missing-module-
    docstring)
simplecaesar.py:5:0: C0103: Constant name "shift" doesn't conform to
    UPPER_CASE naming style (invalid-name)
simplecaesar.py:9:0: C0103: Constant name "encoded" doesn't conform to
    UPPER_CASE naming style (invalid-name)
simplecaesar.py:13:12: C0103: Constant name "encoded" doesn't conform to
    UPPER_CASE naming style (invalid-name)

-----------------------------------
Your code has been rated at 7.37/10
```

Výpis 2.1: Ukázka výstupu nástroje Pylint [6]

Ačkoliv je Pylint vzhledem ke svému zaměření na jazyk Python pro účely této práce nevhodný, tak posloužil jako zdroj inspirace chtěných vlastností tvořeného nástroje.

CodeChecker

CodeChecker[8] je nástroj v jazyce Python pro statickou analýzu postavený nad sadou nástrojů LLVM/Clang Static Analyzer. Skládá se ze dvou částí.

1. **Terminálová aplikace** sloužící ke spouštění Clang-Tidy a Clang Static Analyzer kontrol s možností HTML vizualizace.

2. **Webová aplikace** sloužící jako úložiště hlášení z kontrol, případně k jejich prohlížení a vizualizaci.

Pro cíl této práce je důležitější terminálová aplikace. Ta využívá již vytvořených kontrol pro Clang-Tidy případně Clang Static Analyzer a ve své podstatě pouze volá tyto dva nástroje s určitou zadanou konfigurací a zpracovává její výstupy. Kromě již definovaných kontrol se také nabízí tvorba zcela nových za využití knihoven Clang. Zde se tak otevírá cesta pro tvorbu vlastních kontrol na míru potřebám projektům FITu. Avšak po dalším zkoumání vlastností a možností CodeCheckeru jsem objevil zásadní nedostatky, které jsou dle mého v rozporu se zadáním práce. Jmenovitě to jsou:

[6] https://www.pylint.org/
[7] https://www.python.org/dev/peps/pep-0008/
[8] https://codechecker.readthedocs.io

- Neobsahuje možnost hodnocení kódu (porušení bodu č. 2 zadání - hodnocení kódu)
- Nutnost existence Makefile projektu pro spuštění nástroje (porušení bodu č. 3 zadání - jednoduchost použití)
- Nemožnost parametrizace kontrol nad rámec již definovaných možností (např. počet povolených výskytů)

Jelikož je výsledný program primárně cílen na studenty předmětu Základy programování, je nutno uvažovat pouze základní znalosti ovládání počítače a nulové z oblasti programování. Z toho důvodu nelze dost dobře požadovat po studentech tvorbu Makefile souborů pro projekty, stejně tak operování s docker kontejnerem nutným pro fungování CodeCheckeru. Nemožnost hodnocení kódu pak vidím jako zásadní nedostatek a porušení zadání práce.

Ačkoliv je CodeChecker pro realizaci cílů této práce nevhodný, tak jsem se rozhodl vycházet z jeho konceptu zastřešení nástrojů Clang-Tidy potažmo Clang Static Analyzer. Nejvíce omezující částí CodeCheckeru je totiž právě samotný zastřešovací program (chybějící hodnocení, parametrizace, spouštění, atd.). Implementace vlastních kontrol Clang-Tidy by totiž proběhla stejně jak při využití CodeCheckeru, tak i při vlastní implementaci zastřešujícího programu.

2.2 Statická programová analýza

Statická programová analýza (dále jen zkráceně statická analýza) je přístup, který zkoumá chování systému (systémem rozumíme buďto reálný program anebo model), primárně s využitím jeho zdrojových kódů, aniž by se snažil zdrojový kód provádět anebo ho aspoň neprovádí v původní sémantice.[13]

Statická analýza se začala využívat už v šedesátých letech 20. století v překladačích programovacích jazyků, konkrétně v jejich optimalizační části. V dnešní době se statická analýza využívá též v nástrojích pro vyhledávání chyb v programech. Příkladem takovéhoto nástroje může být např. SpotBugs[9] pro jazyk Java. Dalším příkladem užití statické analýzy mohou být verifikační nástroje, např. Frama-C[10] pro jazyk C. Nebo může být statická analýza využita přímo jako integrální část vývojových prostředí a to pro podporu ladění či refaktorizace kódu[8], např. Microsoft Code analysis for C/C++.

Statickou analýzou lze například provádět analýzy typu: [13]

- vyhledávání chybových vzorů
- analýza toku dat
- analýza založená na omezeních
- analýza typových systémů

Mezi hlavní přednosti statické analýzy patří: [13]

- zvládá velmi rozsáhlé systémy
- častokrát nepotřebuje model okolí (vstup/výstup, knihovny, jiné moduly)

[9] https://spotbugs.github.io/
[10] https://frama-c.com/

- vysoká míra automatizace (jestliže není zkoumaná vlastnost příliš specifická)

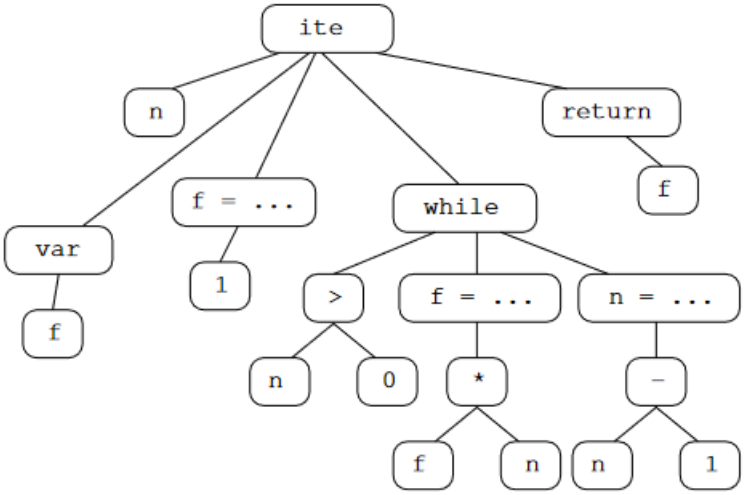
Naopak mezi její nedostatky spadá: [13]

- může vytvářet velké množství falešných hlášeních (tzv. false alarms), to jest označování za chybu i validní části kódu
- velmi úzce specializované analýzy (malé obecné užití, nicméně často se jedná o důsledek snahy o omezení falešných hlášení)
- již z principu má omezenou použitelnost pro některé specifické vlastnosti (ne vše lze vyjádřit chybovým vzorem, např. chyba typu uváznutí)

Statická analýza využívá několik druhů abstrakcí kódu. Nejčastěji jmenovitě: abstraktních syntaktických stromů, grafů toku řízení a grafů volání.

2.2.1 Abstraktní syntaktický strom

Abstraktní syntaktický strom (anglicky abstract syntax tree; AST) je abstrakcí kódu vycházející z derivačního stromu. Na rozdíl od něj však zahazuje všechny neterminály které nejsou potřeba k pochopení struktury. Jedná se tak například vynechání závorek, které jsou realizovány již samotnou stromovou strukturou nebo například realizace podmínek skrze tři separátní uzly (podmínka, pravda větev, nepravda větev). [3]



Obrázek 2.1: AST pro cyklus while [8]

AST se využívají převážně u typové analýzy (angl. type analysis), analýzy toku řízení (angl. control flow analysis), nebo u analýzy ukazatelů (angl. pointer analysis).[8]

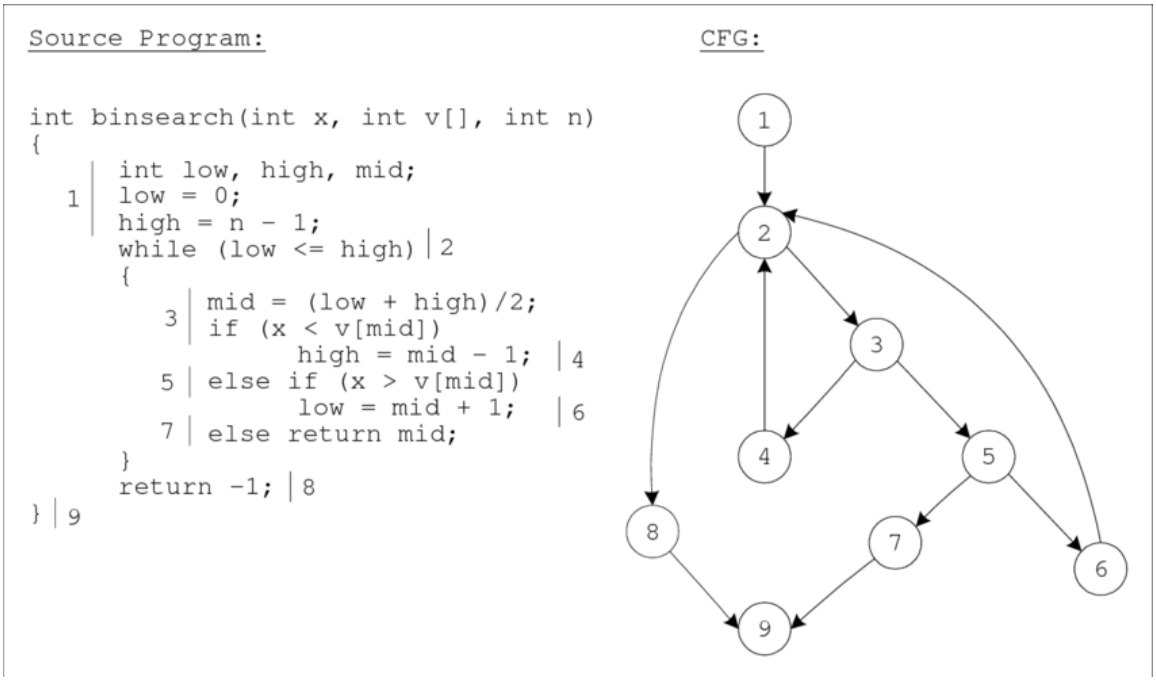
2.2.2 Graf toku řízení

Graf toku řízení (Control Flow Grpah, dále CFG) modeluje tok řízení programu. CFG je orientovaný graf, kde $G = (N, E)$. Každý uzel $n \in N$ odpovídá základnímu bloku [2].

Základní blok (basic block) je posloupnost maximálního počtu příkazů, pro které platí, že: vstupní bod (řízení) je na prvním příkazu, výstupní bod je na posledním příkazu a příkazy se provádí vždy sekvenčně v pořadí dané posloupností. Základní bod může obsahovat

pro větvení (skokové instrukce) pouze na konci (předává řízení jinému základnímu bloku) [10].

Řízení vždy vstupuje do bloku na jeho první operaci a opouští jej na jeho poslední operaci. Každá hrana $e = (n_i, n_j)$ koresponduje s možných přechodem řízení z bloku n_i do bloku n_j [2].



Obrázek 2.2: Příklad zdrojového kódu a k němu korespondujícímu CFG.[1]

Při svém zkoumání možností, jak získat graf toku řízení ke zpracování jsem dospěl ke třem potencionálním způsobům.

Výstup překladače

Překladače GCC a Clang mají již v sobě zahrnuty části pro statickou analýzu a to z důvodu optimalizací překládaného kódu. Tyto části jsou pak po zadání vhodných parametrů schopné vygenerovat CFG i pro uživatele [10].

```
$ gcc $(CFLAGS) -c -fdump-tree-cfg SOURCE.c
```

Výpis 2.2: Generování CFG překladačem GCC [10]

```
$ clang \
  -cc1 -analyze -analyzer-checker=debug.DumpCFG \
  $(CFLAGS) \
  -I/usr/include \
  -I/usr/lib/clang/<verze_clang>/include \
  SOURCE.c
```

Výpis 2.3: Generování CFG překladačem Clang [10]

Tyto vygenerované CFG jsou ve formátu, s nimiž překladače operují během překladu při analytické/optimalizační činnosti.

```
[B2 (ENTRY)]
  Succs (1): B1

[B1]
  1: printf
  2: [B1.1] (ImplicitCastExpr, FunctionToPointerDecay, int (*)(const char
   *, ...))
  3: "Hello, World!"
  4: [B1.3] (ImplicitCastExpr, ArrayToPointerDecay, char *)
  5: [B1.4] (ImplicitCastExpr, NoOp, const char *)
  6: [B1.2]([B1.5])
  7: 0
  8: return [B1.7];
  Preds (1): B2
  Succs (1): B0

[B0 (EXIT)]
  Preds (1): B1
```

Výpis 2.4: CFG programu `hello_world` vygenerované překladačem Clang

Bohužel kvůli nedostatečné dokumentaci a nutnosti tvorby vlastního parseru (zpracovávače) těchto výstupů, jenž nebyl z časových a rozsahových důvodů přijatelný, byla tato možnost po konzultaci s vedoucím práce zamítnuta.

LLVM mezikód

Další možností je využití LLVM mezikódu, což je mezikód do kterého Clang překládá zdrojové soubory v jazyce C, načež LLVM přeloží mezikód na cílovou architekturu uživatele. Knihovny LLVM totiž poskytují programové rozhraní pro manipulaci, analýzu a extrakci mezikódu.

Po konzultaci s vedoucím práce byl tento postup z důvodu svého značného rozsahu zavržen.

LibClang

LibClang[11] je vysokoúrovňové stabilní rozhraní ke Clangu, psané v jazyce C. Ačkoliv jsou typickým případem užití vysokoúrovňové abstrakce nad AST, tak LibClang poskytuje i metody pro práci s CFG. Jmenovitě např. tvorba CFG z kódu, procházení skrze iterátory či případné zjištění jeho velikosti či linearity.

Zásadní nevýhodou pro účely této práce je však velice omezená dokumentace CFG částí knihovny, bez jakéhokoliv manuálu či i ukázek použití.

Vzhledem k menšinovému zastoupení kontrol vyžadujících analýzy CFG, bylo po konzultaci s vedoucím práce rozhodnuto zaměřit se na početněji zastoupené kontroly využívající AST, jenž je mnohem lépe dokumentován než CFG. Avšak nabízí se zde možnost pokračování další prací.

[11] https://clang.llvm.org/doxygen/group__CINDEX.html

2.3 Clang-Tidy

Clang-Tidy je terminálový nástroj typu linter pro jazyky C/C++/Objective-C založený na LibTooling rozhraní Clangu[12], patřícího pod projekt LLVM[13]. Jeho účel spočívá v poskytnutí rozsáhlého aplikačního rámce (angl. framework) pro diagnostiku a opravu typických programátorských chyb, porušování kódovacích stylů, nesprávné užívání rozhraní nebo chyby, které mohou být vydedukovány skrze statickou analýzu.[12]

Clang-Tidy se skládá z modulů z nichž každý obsahuje sadu kontrol (angl. checks). Každý modul představuje určitý sledovaný aspekt. Mezi dostupné moduly patří např. readability (kontroly zaměřené na čitelnost kódu), performance (kontroly sledující programové konstrukce ovlivňující výkon) nebo třeba i konkrétní kódovací konvence (např. google, linuxkernel, llvm, ...). Nad rámec samotných kontrol pro Clang-Tidy, umožňuje nástroj i spouštět kontroly realizované pro Clang Static Analyzer, čímž nadále rozšiřuje svůj možný dosah.

Clang-Tidy má modulární podobu, čili je možné rozšiřovat Clang-Tidy o nové moduly, kontroly, ale i samotné AST vzory.

2.3.1 Kontroly a vzory

Kontroly v Clang-Tidy pracují na základě tzv. AST vzorů (anglicky AST matchers), kdy každá kontrole je svázaná s určitým AST vzorem. Na základě konfigurace Clang-Tidy, tj. jaké kontroly jsou povoleny, se pak vyhledávají ve zkoumaném zdrojovém kódu tyto AST vzory. Při detekci AST vzoru v kódu je vyvoláno zpětné volání (angl. callback) kontroly, která je na něj navázána. Samotný kód kontroly pak zpravidla zajišťuje zaznamenání chyby a informování uživatele o jejím výskytu (výpis na standardní výstup ve formátu: soubor, umístění v kódu, popis chyby, název kontroly, zobrazení konkrétního řádku kódu). Volitelně může kontrola také obsahovat i automatizovanou opravu chyby.

```
user@VM-U20:~/magic_numbers$ clang-tidy -checks=-*,readability-magic-
    numbers* magic_numbers.c --
4 warnings generated.
/home/user/magic_numbers/magic_numbers.c:7:14: warning: 32 is a magic
    number; consider replacing it with a named constant [readability-magic-
    numbers]
  if (cond > 32 && cond < 64 && cond != 55)
             ^
...
```

Výpis 2.5: Ukázka Clang-Tidy výstupu kontroly magických čísel

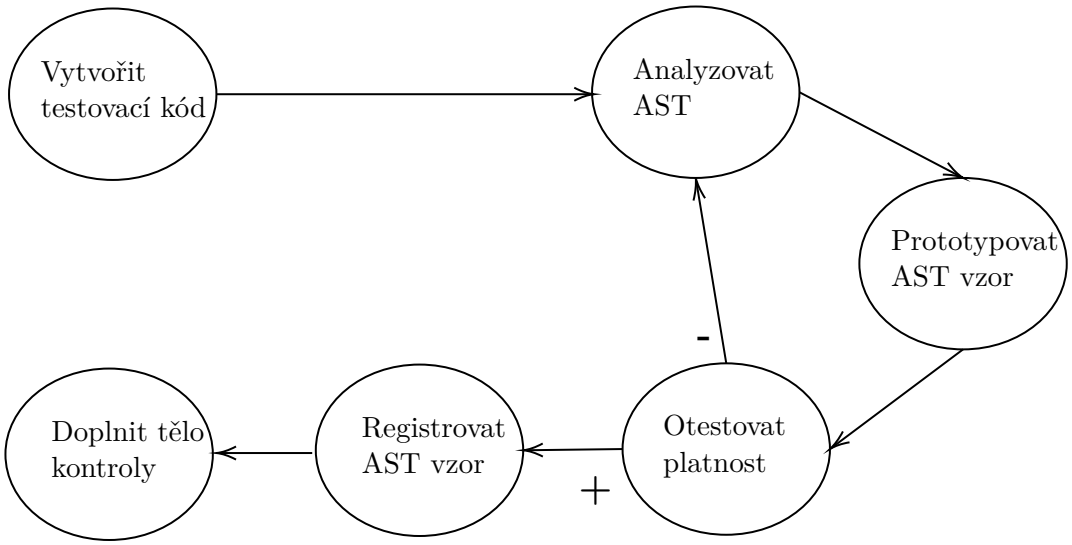
Clang AST vzory využívají knihovny LibASTMatchers. LibASTMatchers poskytuje doménově specifický jazyk k tvorbě predikátů na Clang AST. AST vzory jsou tak predikáty na uzly v AST. Tyto vzory se pak mohou spojovat či zanořovat a vytvářet tak komplexnější vzory, jenž odpovídají specifičtějším uzlům. Clang obsahuje rozsáhlou sadu těchto AST vzorů jako např. *varDecl*, jenž detekuje jakoukoliv deklaraci proměnné. *VarDecl* pak jde kupříkladu nadále zkombinovat s *hasLocalStorage* na *varDecl(hasLocalStorage())*, jenž upřesní vzor pouze na deklarace lokálních proměnných. Podobně lze pokračovat dále a vytvářet komplexnější vzory.

[12] https://clang.llvm.org/
[13] https://llvm.org/

2.3.2 Vývojový cyklus kontroly

Vývoj nové kontroly se dá rozčlenit do několika kroků.

1. Vytvoření testovacího zdrojového kódu, který obsahuje nežádoucí programovou konstrukcí, kterou má kontrola řešit
2. Vygenerování AST daného testovacího kódu (např. pomocí *clang-check –ast-dump source_code*)
3. Analyzování AST a sestavení potřebných predikátů AST vzorů
4. Otestování predikátů na testovacím kódu skrze nástroj clang-query
5. V případě úspěchu pokračovat jinak se navrátit k bodu č. 3
6. Registrovat AST vzory pro danou kontrolu
7. Doplnit tělo kontroly (tj. reakci na detekci registrovaného AST vzoru)



Obrázek 2.3: Vývojový cyklus tvorby kontroly

Prvním krokem je vytvořit testovací kód, jenž bude obsahovat zkoumanou nežádoucí konstrukci, kterou by měla kontrola řešit. Druhým krokem je vygenerovat AST daného testovacího kódu. Doporučeným postupem je využít nástroj clang-check.

```
‘-FunctionDecl 0x1f434f0 </home/user/IBT/init.c:1:1, line:5:1> line:1:5
   main ’int ()’
  ‘-CompoundStmt 0x1f43790 <col:11, line:5:1>
    |-DeclStmt 0x1f43678 <line:2:3, col:12>
    | ‘-VarDecl 0x1f435f0 <col:3, col:11> col:7 i ’int’ cinit
    |   ‘-IntegerLiteral 0x1f43658 <col:11> ’int’ 0
    |-DeclStmt 0x1f43748 <line:3:3, col:17>
    | ‘-VarDecl 0x1f436a8 <col:3, col:13> col:9 f ’float’ cinit
```

```
|   `-ImplicitCastExpr 0x1f43730 <col:13> 'float' <FloatingCast>
|     `-FloatingLiteral 0x1f43710 <col:13> 'double' 3.140000e+00
`-ReturnStmt 0x1f43780 <line:4:3, col:10>
  `-IntegerLiteral 0x1f43760 <col:10> 'int' 0
```

Výpis 2.6: Ukázka části AST, C programu, který inicializuje proměnné typu int a float

Na základě analýzy AST a jeho uzlů se zvolí predikáty AST vzorů, které nejlépe odpovídají struktuře analyzovaného AST. Dalším krokem je otestování těchto predikátů v nástroji clang-query. Clang-query je interaktivní interpret spuštěný nad zkoumaným kódem a dovoluje dotazování se na jeho AST. V praxi to znamená zadání predikátů AST vzorů (různě kombinovaných nebo zanořených) a následné ověření zda dochází ke shodě, tj. že vzor z predikátů popisuje danou konstrukcí. Ověří se také, že nedochází k falešným hlášením, tj. nejsou označeny i jiné konstrukce, jenž by být neměly.

```
user@VM-U20:~/IBT$ clang-query init.c --
clang-query> match varDecl(hasType(isInteger()))

Match #1:

/home/user/IBT/init.c:2:3: note: "root" binds here
  int i = 0;
  ^~~~~~~~
1 match.
```

Výpis 2.7: Ukázka clang-query a predikátů jenž označují proměnné typu int

Pokud nedochází ke shodě nebo jsou za shodu označovány i konstrukce, které by neměly být označeny, tak je třeba návrat ke kroku č. 3 a znovu analyzovat AST a upravit použité predikáty. Jinak se pokračuje dále.

Nyní už zbývá jenom ve zdrojových kódech kontroly přidat (registrovat) predikáty čímž se vytvoří AST vzor, který daná kontrola bude hledat. Poté se již jen dokončí tělo kontroly tj. doplnění reakce kontroly na detekci registrovaného AST vzoru.

Kapitola 3

Návrh řešení

Návrh řešení uvažuje program, jenž bude zastřešovat práci nástroje Clang-Tidy, kdy na základě vstupní konfigurace a výstupu z Clang-Tidy, bude hodnotit zkoumaný zdrojový kód, následně pak vhodně informovat uživatele a poskytnout výstupní shrnutí/zpráv do samostatného souboru. Jedním z hlavních cílů programu je zjednodušit používání a obsluhu oproti použití samostatného Clang-Tidy.

3.1 Zastřešení

Zastřešující program by měl umožňovat analýzu jak jednoho zdrojového souboru, tak i většího množství najednou a to formou dávkového zpracování. Program by měl dále umožňovat přijímat konfigurační soubor na vstupu, případně při jeho absenci se řídit globální konfigurací. V neposlední řadě by pak aplikace měla umožňovat i tichý režim (angl. quiet), kdy aplikace netiskne žádné informace na standardní výstup, ale jenom generuje výstupní zprávy. Tento režim je motivován použitím dávkového zpracování s cílem získat pouze výstupní zprávy, za účelem dalšího statistického zpracování.

3.1.1 Hodnocení

Pro bodové hodnocení zkoumaných zdrojových kódu jsem se rozhodl převzít hodnotící vzorec z nástroje Pylint (viz 2.1.2). Ten ve svém původním tvaru vypadá následovně: [9]

$$10.0 - ((float(5 * error + warning + refactor + convention)/statement) * 10)$$

kde:

$error$ - počet syntaktických chyb,

$warning$ - počet varování,

$refactor$ - počet konstrukcí u nichž byla doporučena refaktorizace,

$convention$ - počet porušení programových konvencí,

$statement$ - počet příkazů analyzovaného programu,

Poznámka: $float$ ve vzorci představuje pouze datový typ.

Vzorec ukazuje, že Pylint hodnotí kromě porušení konvencí nebo doporučení k refaktorizaci, též syntaktické chyby a varování. Ty nejsou pro potřeby hodnocení úhlednosti důležité, proto jsem se rozhodl vytvořit upravenou verzi. Oproti původnímu vzorci jsem se rozhodl zachovat pouze jednu kategorii chyb, nazvěme ji chyby způsobující neúhlednost a přidat k ní různé úrovně závažnosti chyb, podobně jako u původního hodnocení syntaktických chyb. V prvotní implementaci se uvažuje tříúrovňová závažnost, tj. nízká, střední a vysoká závažnost chyby krát suma výskytů jejich úrovně. Po těchto úpravách pak vypadá vzorec následovně:

$$body_{max} - (((vaha_{vys} * sum_{vys} + vaha_{str} * sum_{str} + vaha_{niz} * sum_{niz})/prikazy) * body_{max})$$

kde:

$body_{max}$ - maximální počet bodů, jenž může být udělen,

$vaha_{vys}$ - konstanta určující váhu chyb s vysokou závažností,

sum_{vys} - počet výskytů chyb s vysokou závažností,

$vaha_{str}$ - konstanta určující váhu chyb s střední závažností,

sum_{str} - počet výskytů chyb se střední závažností,

$vaha_{niz}$ - konstanta určující váhu chyb s nízkou závažností,

sum_{niz} - počet výskytů chyb s nízkou závažností,

$prikazy$ - počet příkazů analyzovaného programu.

Stejně jako v původním vzorci se i zde v celkovém hodnocení zohledňuje počet řádků vstupního zdrojového kódu. A to konkrétně poměrem počtu chyb včetně jejich vah vůči počtu příkazů ve zkoumaném zdrojovém kódu vstupního souboru. Motivace za tímto krokem spočívala odlišit hodnocení zdrojových kódů s různým rozsahem.

Uvažujme příklad dvou programů, programu A a programu B. Program A má rozsah 50 příkazů, kdežto program B má rozsah 500 příkazů. Oba programy obsahují 25 totožných chyb způsobujících neúhlednost. Kdežto u programu B to představuje chybovost 5%, tak u programu A už to činí 50%. Program A by tak měl mít nižší hodnocení než program B, když je polovina jeho kódu neúhledná oproti chybovosti v jednotkách procent u programu B.

3.1.2 Konfigurace aplikace

Struktura konfiguračního souboru uvažuje strukturu: seznam povolených kontrol, váha těchto kontrol při vyhodnocování případně nastavení parametrů kontrol, je-li třeba.

Aplikace uvažuje několik úrovní konfigurací s různým stupněm priority. Na nejvyšší úrovni je konfigurace vložená na vstupu aplikace, neboli lokální konfigurace. Tato lokální konfigurace bude akceptovaná vždy nehledě na ostatní.

Pokud není určena lokální konfigurace, operuje aplikace s globální konfigurací uloženou v konfiguračních souborech operačního systému. Tato konfigurace je vytvořena při instalaci aplikace a ve výchozím stavu obsahuje nastavení kontrol takovým způsobem, jaký by měl být platný u většiny projektů. Tuto konfiguraci je však možno nadále upravovat dle potřeby.

Na poslední úrovni je konfigurace obsažená v samotné aplikaci. Ta obsahuje pouze redukovanou sadu kontrol, které jsou všeobecně platné za jakýchkoliv podmínek. Neobsahuje tak kontroly vynucující např. určitý styl formátování, ale naopak obsahuje kontroly spíše funkčního charakteru (např. detekce magických čísel, redundantních výrazů, apod.). Tato konfigurace je natvrdo zakódovaná a představuje poslední možné nastavení pro chod aplikace.

3.1.3 Výstupní zpráva

Výstupní zpráva představuje souhrn výsledků aplikace. To jest výpis spuštěných kontrol, váhy daných kontrol, počet výskytů dané chyby, počet příkazů ve zdrojovém kódu, vzorec využitý pro výpočet bodů a samotné bodové ohodnocení zkoumaného souboru.

Zpráva jako taková uvažuje formát souboru pro serializaci strukturovaných dat jako např. JSON, XML, YAML, aj. Motivace za tímto rozhodnutím je mít výstup ve formátu, jenž je dobře strojově zpracovatelný a tím i ponechat možnost dalšího budoucího rozšiřování aplikace. Tím může být např. vizualizace výsledků, tvorba statistik na jejich základě apod.

3.2 Clang Tidy

V následující části bude popsán návrh využití nástroje Clang-Tidy a jeho možností.

3.2.1 Kontroly a vzory

Na základě konzultací s vedoucím práce o požadavcích na kontroly bylo vyhodnoceno 15 kontrol, již vestavěných v Clang-Tidy, jako vhodné pro účely kontroly úhlednosti. Případné další požadavky na kontroly by byly realizovány formou nových kontrol pro Clang-Tidy, případně Clang-Static-Analyzer. Přehled všech použitých kontrol včetně jejich stručných popisů se nachází v příloze A. Příkladem těchto kontrol jsou např. `readability-magic-numbers` a `readability-function-size`.

readability-magic-numbers

Kontrola `readability-magic-numbers`[1] slouží k detekci magických čísel. Magická čísla jsou celočíselné nebo desetinné literály, které jsou natvrdo zapsány v kódu a nepochází z konstant nebo symbolů.

```
int bit = 32; // priklad inicializace promenne magickym cislem
if (bit == 32) { // priklad magickeho cisla v podmince
  return 0; // priklad magickeho cisla jako navratove hodnoty
}
else {
  return -1; // priklad magickeho cisla jako navratove hodnoty
}
```

Výpis 3.1: Příklad magických čísel v jazyce C

Tato kontrola realizuje požadavek vyskytující se v mnoha kódovacích konvencích a stylech a to nahradit mnohdy nicneříkající číselné konstanty těmi pojmenovanými. Pojmenované

[1] https://clang.llvm.org/extra/clang-tidy/checks/readability-magic-numbers.html

konstanty jsou na první pohled čitelnější, jelikož zpravidla již ve svém názvu obsahují i svůj význam.

```
int bit = BIT32;
if (bit == BIT32) {
  return EXIT_SUCCESS;
}
else {
  return EXIT_FAILURE;
}
```

Výpis 3.2: Opravený příklad magických čísel v jazyce C

Kontrola nabízí také parametrizaci. Přepínač `IgnoredIntegerValues` dovoluje nastavit množinu celočíselných hodnot ignorovaných kontrolou. Ve svém výchozím nastavení ignoruje hodnoty 1, 2, 3, 4. Nulu ignoruje vždy, jelikož se jedná o velice používanou hodnotu a to např. pro inicializování čítačů či řídících proměnných cyklů. Další z parametrů je třeba `IgnorePowersOf2IntegerValues`, který bude ignorovat všechny celočíselné mocniny dvou. V obdobném duchu pak následují i parametry pro desetinné hodnoty.

Kapitola 4

Implementace

Jako cílová platforma byl zvolen systém Linux, přesněji distribuce Ubuntu ve vydání 20.04 LTS.

4.1 llvm-project

Pro provoz upraveného Clang-Tidy o nové moduly a kontroly je zapotřebí stažení původních zdrojových kódů a s každou novou kontrolou pak znovu přeložit aplikaci.

Zdrojové kódy jsou veřejně dostupné v rámci repozitáře llvm-project[1] na službě Github. Repozitář obsahuje kromě samotného LLVM i zdrojové kódy všech k němu náležících projektů jako např. Clang, libc++ a další.

4.1.1 Modul FIT

Jednou z částí implementace bylo vytvoření nového modulu, který by obsahoval nové vlastní kontroly nad rámec již dodávaných s Clang-Tidy. Přidání nového modulu (pojmenovaného FIT), spočívalo ve vytvoření stejnojmenného adresáře uvnitř adresáře Clang-Tidy. Tento nový adresář pak obsahuje soubory XTidyModule.cpp a CMakeList.txt. XTidyModule.cpp (kdy za X dosadit jméno modulu, zde FIT), obsahující registraci modulu a továren pro tvorbu kontrol pod něj spadající. CMakeList.txt pro kompilaci daného modulu a kontrol. Mimo adresář s modulem je dále třeba upravit CMakeList.txt samotného Clang-Tidy a to právě přidáním modulu a jeho adresáře. Posledním krokem pak je již jenom přidat do linkeru údaj o místě, kde se nový modul nachází. Poté stačí již jenom znovu přeložit Clang-Tidy a je možno vytvářet kontroly pro nový vlastní modul.

4.1.2 Vlastní kontroly

Jako první kontrola pro potvrzení funkčnosti byla vybrána kontrola pojmenovaná `fit-too-big-local-vars`. Jedná se o kontrolu, jenž má kontrolovat velikost lokálních proměnných. Motivací je odradit začínající programátory od výhradního spoléhání na alokování paměti na zásobníku (angl. stack) a naopak podpořit užívání alokace paměti na hromadě (angl. heap), tj. využívat funkcí `malloc()` a `free()`.

Prvním krokem je využití skriptu dodávaného s Clang-Tidy a to `add_new_check.py` pro vytvoření zdrojových souborů kontroly a jejich zanesení do potřebných CMakeList.txt souborů, pro zajištění zkompilování. Nad rámec nutného minima vytváří zároveň skript i

[1] https://github.com/llvm/llvm-project

soubor pro dokumentaci a také soubor s testovacím případ nové kontroly pro lit (LLVM integrated tester) testování.

Dalším krokem je registrování AST vzoru pro tuto kontrolu. K tomu se používají metody `registerMatchers`, respektive metody `addMatcher`, které se jako parametr předají právě predikáty popisující AST vzor.

```
void TooBigLocalVarsCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(varDecl(hasLocalStorage()).bind("too_big_var"), this);
}
```

Výpis 4.1: Ukázka kódu pro registrování AST vzoru (`TooBigLocalVarsCheck.cpp`)

AST vzor se skládá z predikátů `varDecl`, který označí jakoukoliv deklaraci proměnné a do něj je zanořen `hasLocalStorage`, který výběr zúží pouze na lokální proměnné. Poté se již jen zvolí interní označení v případě nalezení zde `"too_big_var"`.

Poté je třeba specifikovat samotnou náplň kontroly a to doplněním šablony metody `check`.

```
void TooBigLocalVarsCheck::check(const MatchFinder::MatchResult &Result) {
  if (const auto *MatchedDecl = Result.Nodes.getNodeAs<VarDecl>("
      too_big_var")) {
    auto TypeInfo = MatchedDecl->getASTContext().getTypeInfo(MatchedDecl->
        getType());
    if (TypeInfo.Width < MaxSize)
      return;
    diag(MatchedDecl->getLocation(), "variable %0 is too big")
        << MatchedDecl;
  }
}
```

Výpis 4.2: Ukázka kódu pro kontroly (`TooBigLocalVarsCheck.cpp`)

Nejprve se vyfiltruje AST uzel, který byl detekován AST vzorem. Poté se zjistí velikost daného uzlu v bytech a to za pomocí struktury `TypeInfo` a jejího členu `Width`. Pokud je velikost uzlu menší než je maximální povolená (obsažena v proměnné `MaxSize`, výchozí hodnota je 8MB), pak zkoumaná proměnná splňuje maximální povolenou velikost proměnné a kontrola uzlu se ukončí. Jestliže je větší pak nesplňuje stanovenou maximální velikost a uživatel je informován na standardní výstup a to ve formátu: `umístění chyby, popis chyby, název kontroly a konkrétní řádek kódu`, například.

```
/home/user/IBT/too_big_loc_vars.c:21:8: warning: variable 'longArr' is too
    big [fit-too-big-local-vars]
  long longArr[500000];
       ^
/home/user/too_big_loc_vars.c:24:14: warning: variable 'b1' is too big [fit
    -too-big-local-vars]
  struct big b1;
             ^
```

Výpis 4.3: Ukázka výstupu kontroly fit-too-big-local-vars

Posledním volitelným krokem je přidání parametrizace kontroly. V případě kontroly `fit-too-big-local-vars` jsem se rozhodl zahrnout možnost nastavit maximální povolenou

velikost proměnné. Parametry kontroly se zavedou tak, že se přepíše (angl. overide) virtuální metody storeOptions třídy ClangTidyCheck a stanoví tak jednotlivé parametry. Poté již stačí jenom přidat inicializaci parametru do konstruktoru kontroly. Jako první se vybírá hodnota specifikovaná v konfiguraci kontroly jinak se vybere výchozí hodnota 8 MB.

```
void TooBigLocalVarsCheck::storeOptions(ClangTidyOptions::OptionMap &Opts)
    {
  Options.store(Opts, "MaxSize", MaxSize);
}

TooBigLocalVarsCheck::TooBigLocalVarsCheck(StringRef Name, ClangTidyContext
     *Context)
      : ClangTidyCheck(Name, Context),
        MaxSize(Options.get("MaxSize", 8000000)) {}
```

Výpis 4.4: Kód realizující parametrizaci kontroly fit-too-big-local-vars

Jakmile je kontrola dokončena přistupuje se k vytvoření lit[2] testu pro ověření funkčnosti kontroly. Tyto testy fungují na základě ukázkového kódu, který by měl spustit kontrolu a následně regulárního výrazu popisující generované chybové hlášení.

```
// RUN: %check_clang_tidy %s fit-too-big-local-vars %t --
void foo() {
  long longArr[500000];
}
// CHECK-MESSAGES: :[[@LINE-2]]:8: warning: variable 'longArr' is too big [
    fit-too-big-local-vars]
```

Výpis 4.5: Ukázka lit testu pro základní chování kontroly `fit-too-big-local-vars`

Vývojář může vytvořit libovolné množství lit testů nad různými zdrojovými kódy. Často se jimi ověřují i veškeré možné kombinace při využití parametrizace kontrol jako např. zde při využití parametru `MaxSize`.

```
// RUN: %check_clang_tidy %s fit-too-big-local-vars %t -- \
// RUN: -config='{CheckOptions: [ \
// RUN: {key: fit-too-big-local-vars.MaxSize, value: 16}, \
// RUN: ]}'

void foo() {
  int a = 10;
}
// CHECK-MESSAGES: :[[@LINE-2]]:7: warning: variable 'a' is too big [fit-
    too-big-local-vars]
```

Výpis 4.6: Ukázka lit testu pro parametrizovanou kontrolu `fit-too-big-local-vars`

[2] https://llvm.org/docs/CommandGuide/lit.html

4.1.3 Upravené kontroly

Další dvě kontroly spadající pod modul FIT jsou upravené verze vestavěných kontrol `cppcoreguidelines-avoid-goto`[3] a `cppcoreguidelines-macro-usage`[4]. Ty sice realizují kódovací styl CppCoreGuidelines[5] od ISOCPP pro jazyk C++, přesto jejich princip, tj. omezení užívání nepodmíněných skoků a užívání maker, byl po konzultacích o požadavcích vyhodnocen jako platný a přínosný i pro programy v jazyce C. Jejich úpravy tak spočívaly v odstranění omezení pro operování pouze nad zdrojovými kódy v jazyce C++. Odstranění sledování ryze C++ konstrukcí jako například rozsahové cykly (angl. range-based loops) nebo třeba `constexpr`. A v neposlední řadě úprava zpráv s doporučeními jak chyby napravit.

4.2 Zastřešovací program

Jako programovací jazyk pro tvorbu zastřešovacího programu byl zvolen jazyk C++ a to konkrétně standardu C++17. Hlavní důvody pro jeho výběr bylo možné využití LibTooling případně LibClang pro přístup k AST a dalším možnostem knihoven Clang pro statickou analýzu. Mezi další důvody patří také výkonnost jazyka C++, možnost více procesového a vícevláknového zpracování, plná kontrola na správou paměti atd.

Program `revisor`, který je realizovaný touto prací, je implementován jako terminálové aplikace.

Aplikace očekává na vstupu zdrojový (přepínač -s) nebo dávkový (přepínač -b) soubor ke zkoumání. Dávkovým souborem je zde myšleno soubor obsahující seznam zdrojových souborů ke kontrole.

Volitelně pak aplikace přijímá konfigurační soubor, obsahující parametry nastavující běh aplikace (seznam spouštěných kontrol, jejich váhy v hodnocení atd.). V případě jeho neuvedení se pak pokračuje na další stupeň v hierarchii konfigurací (více viz 4.2.1).

4.2.1 Konfigurace

Jako formát konfiguračních souborů jsem se rozhodl zvolit formát YAML (YAML Ain't Markup Language). Hlavním důvodem je, že Clang-Tidy interně využívá formát YAML pro svou konfiguraci. Formát YAML je využíván jak v lokálním konfiguračním souboru `.clang-tidy`, tak i při využití přepínače `-config`.

Jako další pozitivum vyplývající z využití formátu YAML je také jednoduchost formátu. Jednoduchost formátu a s ní spjatá čitelnosti dovoluje i málo zkušenným lidem v oblasti informačních technologií jednoduše chápat obsah souboru, čímž mohou snadno upravovat nebo i tvořit nové konfigurace.

```
static_analysis:
  clang-tidy:
    checks:
      fit-c-avoid-goto:
        weight: 3
      fit-c-macro-usage:
        weight: 2
      readability-function-size:
```

[3] https://clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines-avoid-goto.html
[4] https://clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines-macro-usage.html
[5] http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

```
param:
  LineThreshold: 100
  BranchThreshold: 5
  ParameterThreshold: 5
  NestingThreshold: 20
  VariableThreshold: 30
weight: 1
```

Výpis 4.7: Příklad konfiguračního souboru

Struktura konfigurace začíná na nejvyšší úrovni mapou `static_anylysis`, která má zahrnovat vše potřebné pro nastavení částí využívajících statickou analýzu. Ačkoliv je `static_anylysis` prozatím jedinou mapou na nejvyšší úrovni, počítá se do budoucna s vícero funkčními moduly (využívající např. metod syntaktické analýzy), avšak jednotným konfiguračním souborem. Jako hodnota `static_anylysis` je další mapa tentokráte `clang-tidy`. Ta obsahuje veškeré potřebné nastavení týkající se provozu Clang-Tidy a jeho kontrol. Opět je zde tak otevřena možnost využití dalších nástrojů než jen Clang-Tidy. Každá kontrola představuje vlastní mapu obsahující váhu (angl. `weight`) dané kontroly a případně i parametry kontroly. Očekávaná hodnota váhy kontroly je 0-3, přičemž hodnota 0 není zahrnuta do bodového hodnocení a ve výpisu shrnutí dané kontroly figurují jako nehodnocená doporučení. Zařazeny zde byly hlavně kontroly prosazující určitý styl formátování nebo redundantní konstrukce, které jsou však explicitnější a pro začátečníky srozumitelnější. Mezi tyto kontroly patří např. `readability-redundant-control-flow`[6] nebo `readability-else-after-return`[7]. Jiné hodnoty než výše zmíněné jsou ignorovány a s nimi i kontroly samotné.

yaml-cpp

Pro zpracování YAML souborů byla vybrána knihovna yaml-cpp[8] to ve verzi 0.6.3, jakožto nejnovější v době vývoje. Jedná se o nejrozvinutější řešení pro jazyk C++, které je uváděno i na oficiálních stránkách YAML formátu [15].

Knihovna byla využita jak pro zpracování konfiguračních souborů, tak i pro generování výstupních zpráv.

Z implementačních detailů knihovny, jenž stojí za zmínku a které ovlivňují i chování aplikace je řešení duplicit, respektive neunikátních klíčů u mapování. Ačkoliv standard YAML zakazuje duplicitní záznamy [14] přesto yaml-cpp v současné verzi takový soubor zpracuje. Chování knihovny v tomto případě je pak takové, že uvažuje pouze první takovýto záznam a všechny další redefinice ignoruje.

4.2.2 Výstupní zpráva

Jako formát výstupní zprávy byl stejně jako u vstupní konfigurace zvolen formát YAML. Výstupní zpráva sdílí základní strukturu se vstupní konfigurací a to až po mapu `clang-tidy`. Ta je rozšířena o novou mapu `occurrence` obsahující počet výskytů chyb ve zkoumaném kódu spouštějící danou kontrolu. Mimo `clang-tidy` je mapa `static_analysis` rozšířena o další položky, převážně o metadata o běhu a analyzovaném souboru. Těmi jsou počet

[6] https://clang.llvm.org/extra/clang-tidy/checks/readability-redundant-control-flow.html
[7] https://clang.llvm.org/extra/clang-tidy/checks/readability-else-after-return.html
[8] https://github.com/jbeder/yaml-cpp

řádků zkoumaného kódu (`line_of_code`), vzorec využitý při výpočtu bodů (`formula`), počet udělených bodů (`points`) a počet výskytů jednotlivých chyb (`occurence`).

```
static_analysis:
  line_of_code: 49
  evaluation:
    formula: ThreeLevelPylint
    points: 4.489796
  clang-tidy:
    checks:
      fit-c-avoid-goto:
        occurrence: 0
        weight: 3
      readability-braces-around-statements:
        occurrence: 6
        weight: 2
...
```

Výpis 4.8: Ukázka výstupní zprávy

Název souboru s výstupní zprávou se skládá z názvu zkoumaného souboru ke kterému je přidán sufix `"_report.yaml"`.

4.2.3 Třídy

Program `revisor` se skládá ze tří tříd, jmenovitě `Revisor_config`, `CT_operator` a `Eval`.

Revisor_config

Třída `Revisor_config` řeší veškerou konfiguraci aplikace. Zpracovává přepínače a argumenty z příkazové řádky. Načítá a převádí vstupní YAML konfiguraci do vnitřní reprezentace aplikace. Totéž platí i pro dávkový soubor, je-li přítomen.

Zpracování YAML konfigurace provádí metody `load_config`. Tato metoda zároveň realizuje kaskádu konfigurací (viz 3.1.2), přičemž tato metodu obsahuje právě onu minimální redukovanou sadu kontrol. Povolené kontroly a jejich váhy jsou interně uloženy v asociativním kontejneru `checks_weight` typu `std::map`, přičemž klíč je datového typu `std::string` a k němu náležící hodnota pak typu `int`. V případě parametrizace kontroly v konfiguraci se pak rovněž využívá asociativního kontejneru, zde jménem `checks_params`. Avšak na rozdíl od předchozího je nyní jako datový typ hodnoty další asociativní kontejner, tenkráte klíč i hodnota typu `std::string`. Ten má na pozici klíče název parametru a na pozici hodnoty pak samotnou hodnotu parametru. Tímto je replikována struktura konfigurace.

Zpracování zdrojového i dávkového souboru probíhá metodou `load_files`. Názvy souboru či souborů se uchovávají v kontejneru typu `std::vector`, přičemž datový typ elementů je `std::string`. Soubory jsou pak následně v hlavní smyčce programu postupně vybírány z kontejneru a spouští se nad nimi kontroly definované konfigurací.

CT_operator

Třída `CT_operator` zajišťuje veškerou interakci s nástrojem Clang-Tidy. Na základě konfigurace v třídě `Revisor_config` pak nastavuje Clang-Tidy, spouští jej a ukládá jeho výstup.

Pro využití Clang-Tidy přímo z programu bylo přistoupeno k využití kombinace `fork()`, `exec()` a komunikací za pomoci roury (angl. pipe). Nejdříve je však nutné metodou `prepare_params` naformátovat a připravit všechny vstupní parametry Clang-Tidy, dle vstupní konfigurace. Až poté se přistoupí k zavolání metody `run_ct`.

Metoda `run_ct` nejdříve ustanoví komunikační rouru, která bude využita pro komunikaci Clang-Tidy s `revisor`. Po vytvoření roury se provede duplikace procesu pomocí `fork()`. Potomek následně uzavře svůj konec roury pro čtení. Ponechá pouze zápisový a převede na něj standardní výstup. Vše co tedy proces potomka bude zapisovat na standardní výstup poputuje rourou i k rodičovskému procesu. Poté již potomek jen provede `execlp()` s parametry pro spuštění `clang-tidy`. Tím dojde ke spuštění Clang-Tidy, jehož proces nahradí proces potomka. Rodičovský proces mezitím uzavře svůj konec roury pro zápis. Tím je zajištěna jednosměrná komunikační roura od potomka k rodičovi. Poté již jen probíhá čtení z roury od potomka po bytech a ukládání přečtených dat do textového řetězce. Poté co se již dosáhlo konce čtení, navrací metoda textový řetězec s obsahem běhu `clang-tidy`.

Eval

Třída `Eval` vykonává veškeré zpracování Clang-Tidy výstupu, jeho vyhodnocování, informování uživatele o výsledcích a tvorbu výstupní zprávy.

Metody třídy lze rozdělit na dvě skupiny. Metody zajišťující zjištění počtu chyb pro jednotlivé kontroly a výpočet bodů:

- `count_occurrences` - pro výpočet počtu výskytů jednotlivých chyb
- `count_weight_sum` - pro výpočet počtu chyb pro jednotlivé váhy
- `count_loc` - pro spočítání řádků kódu zkoumaného souboru
- `count_points` - výpočet bodového ohodnocení (viz 3.1.1)

A výstup hodnocení:

- `print_occurrences` - informování uživatele na standardní výstup o výsledcích jednotlivých kontrol
- `print_points` - informování uživatele na standardní výstup o uděleném počtu bodů
- `make_yaml_report` - vytvoření výsledné YAML zprávy

Po spuštění programu `revisor` a po informování o konkrétních chybách (jsou-li přítomny) následuje shrnutí stavu všech kontrol a výpis udělených bodů.

```
--------------------------------------------------------------------------------

Checks result:
................................................................................

Recommendations (not evaluated):

................................................................................

Low severity:

misc-redundant-expression : PASSED
................................................................................

Medium severity:

fit-too-big-local-vars : PASSED
readability-braces-around-statements : FAILED (2x)
readability-magic-numbers : FAILED (1x)
................................................................................

High severity:

bugprone-infinite-loop : PASSED

--------------------------------------------------------------------------------

Rating:
6 points out of 10

********************************************************************************

********************************************************************************
```

Výpis 4.9: Ukázka výstupu `revisor` na standardní výstup

Kapitola 5

Testování

V první fázi probíhalo testování na malých umělých příkladech kódů, které měly za cíl otestovat funkčnost jednotlivých kontrol. V druhé fázi testovaní se pak již přikročilo k testování nad reálnými daty.

5.1 Testování na umělých příkladech

Během průzkumu dostupných kontrol byla vypracována sada zdrojových kódů, jenž měla otestovat funkcionalitu jednotlivých kontrol a sloužit pro následné ověřování funkčnosti v rámci programu `revisor`. Mimo umělé testovací kódy pro jednotlivé kontroly však byly vytvořeny i demonstrační příklady realizující menší programy, avšak se zanesenými chybami, které způsobují neúhlednost. Cílem těchto testů bylo mimo jiné i zjistit, jak se bude měnit hodnocení s postupným opravováním chyb.

Jedním z testovacích programů byl program realizující základní algoritmus pro řazení a to bublinkové řazení (angl. Bubble Sort). Do samotného programu však byla uměle zanesena řada chyb, vždy dvě za jednu váhovou úroveň. V příloze C se nachází plný text zdrojového kódu, použitá konfigurace a i kompletní výpis na standardní výstup.

Zanesené chyby byly pro nízkou úroveň byly

- `misc-unused-parameters`,
- `readability-redundant-declaration`.

Pro střední úroveň

- `bugprone-redundant-branch-condition`,
- `readability-braces-around-statements`.

Pro vysokou úroveň

- `bugprone-infinite-loop`,
- `fit-c-avoid-goto`

Tabulka 5.1: Bodové ohodnocení postupně opravovaného bubble_sort.c

Chyby	Bodové ohodnocení
2N+2S+2V	6.36364
2N+2S	7.93103
2N	9.33333
N+S+V	8.23529
N+S	9.0
N	9.66667

Pozn. kde N = nízká závažnost, S = střední závažnost a V = vysoká závažnost.

Po prozkoumání původního kódu ohodnotil `revisor` program `bubble_sort.c` 6.36364 body z 10. Po odstranění chyb s vysokou závažností a ponechání střední a nízké bylo uděleno hodnocení 7.93103 bodů. Odstraněním chyb s vysokou závažností tak bylo množství ztracených bodů redukované téměř na polovinu. Kdežto oprava chyb s nízkou úrovní závažnosti vylepšuje hodnocení minimálně. Reálné chování tedy odpovídá tomu očekávanému.

5.2 Testování na reálných datech

Testování na reálných datech je rozděleno do dvou částí a to konkrétně: testování nad studentskými projekty (IZP) a testování komplexního programu (IFJ)

5.2.1 Studentské projekty IZP

Jelikož je cílem této práce vytvořit program na hodnocení úhlednosti, který by mohl být nasazen v předmětu Základy programování (IZP), byla mi po dohodě s vedoucím práce poskytnuta sada anonymizovaných projektů právě z předmětu IZP a to za školní rok 2019/2020. Pro demonstraci byly využity sady z prvního a třetího projektu.

Testovací sada prvního projektu obsahuje 30 zdrojových kódů. Testovací sada třetího projektu obsahuje 28 zdrojových kódů, (dva studenti neodevzdali). Zdrojové kódy pocházejících od stejných autorů. Jako vstupní konfigurace byla využita výchozí globální konfigurace (detaily nastavení viz příloha B).

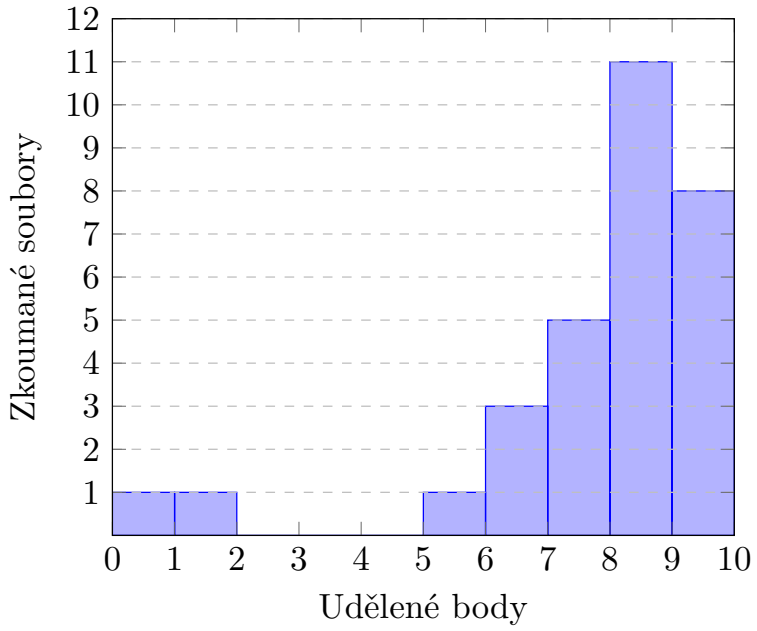
První projekt IZP

Cílem prvního projektu bylo vytvořit program, který bude implementovat základní operace tabulkových procesorů. Vstupem programu budou textová data, zadání operací bude prostřednictvím argumentů příkazové řádky a svůj výsledek bude program vypisovat na výstup.

Tabulka 5.2: Výsledky testování prvního projektu IZP (30 souborů)

Kontrola	Počet výskytů	Průměrné množství
Nehodnocené kontroly (doporučení):		
readability-else-after-return	17x	1.0667
readability-misleading-indentation	6x	0.4
readability-redundant-control-flow	2x	0.1
Nízká úroveň		
misc-redundant-expression	1x	0.0333
misc-unused-parameters	1x	0.0333
readability-function-size	1x	0.0333
readability-redundant-declaration	0x	0
readability-redundant-function-ptr-dereference	0x	0
readability-redundant-preprocessor	0x	0
Střední úroveň		
bugprone-redundant-branch-condition	0x	0
fit-c-macro-usage	11x	0.8667
fit-too-big-local-vars	0x	0
readability-braces-around-statements	24x	7.4333
readability-magic-numbers	29x	9.3333
Vysoká úroveň		
bugprone-infinite-loop	0x	0
fit-c-avoid-goto	0x	0

Ze shrnutí vyplývá, že zdaleka nejčastěji se vyskytovaly chyby typu magická čísla a chybějící složené závorky pro těla podmínek a cyklů. V případě magických čísel tomu tak bylo v 29 ze 30 zkoumaných zdrojových kódu a v případě chybějících závorek pak 24 ze 30. Třetí v pořadí s 11 případy z 30 je využívání maker, což může, ale nemusí působit problémy.



Obrázek 5.1: Histogram uděleného bodového hodnocení prvního projektu IZP

Nadpoloviční většina projektů získala nadprůměrné hodnocení. Hodnocení kolem 5 bodů obdržely projekty u nichž byl detekován takový počet chyb, který rozsahově odpovídal třetině jejich celkového kódu. Tento trend pokračuje u projektu s hodnocením 1.5596, kde podíl chyb způsobujících neúhlednost tvoří již takřka polovinu. Zato u projektu s nulovým bodovým ohodnocením již tvořil neúhledný kód víc jak polovinu celkového kódu.

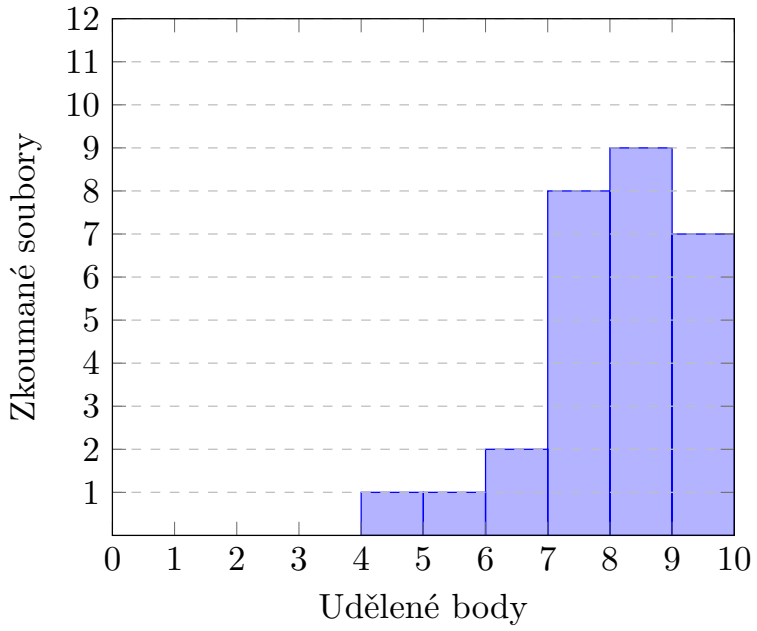
Třetí projekt IZP

Cílem třetího projektu bylo vytvořit program, který v daném bludišti a jeho vstupu najde průchod ven. Bludiště je uloženo v textovém souboru ve formě obdélníkové matice celých čísel. Cílem programu je výpis souřadnic políček bludiště, přes které vede cesta z vchodu bludiště do jeho východu.

Tabulka 5.3: Výsledky testování třetího projektu IZP (28 souborů)

Kontrola	Počet výskytů	Průměrné množství
Nehodnocené kontroly (doporučení):		
readability-else-after-return	24x	6.2143
readability-misleading-indentation	9x	0.8571
readability-redundant-control-flow	5x	0.2857
Nízká úroveň		
misc-redundant-expression	2x	0.0714
misc-unused-parameters	1x	0.0714
readability-function-size	10x	0.4643
readability-redundant-declaration	0x	0
readability-redundant-function-ptr-dereference	0x	0
readability-redundant-preprocessor	0x	0
Střední úroveň		
bugprone-redundant-branch-condition	0x	0
fit-c-macro-usage	10x	3.0357
fit-too-big-local-vars	0x	0
readability-braces-around-statements	23x	26.3214
readability-magic-numbers	25x	5.6071
Vysoká úroveň		
bugprone-infinite-loop	0x	0
fit-c-avoid-goto	0x	0

Z výsledků testování vyplývá, že ačkoliv se průměrný rozsah projektu zhruba zdvojnásobil oproti prvnímu projektu (konkrétně 392 proti 170 řádkům), tak počet výskytů chyb zůstává relativně stejný. Nárůst nastává u kontrol `readability-function-size`, `fit-c-macro-usage` a ve všech nehodnocených. Nárůst u `readability-function-size` způsobilo přesažení maximálního počtu povolených řádků pro funkci případně přesažení limitu větví. Polovina kódů také využívala konstanty deklarované makry. U těchto kódů zásadně ubylo užití magických čísel. Lze tedy u studentů pozorovat rozvíjející snahu o čitelnější kód.



Obrázek 5.2: Histogram uděleného bodového hodnocení třetího projektu IZP

Při porovnání bodových ohodnocení prvního a třetího je zde vidět zlepšení. Nejnižší udělené hodnocení se posunulo z 0 na 4.9217. Celkově 85% všech kódů spadá do horní čtvrtiny hodnocení.

5.2.2 Komplexnější testovací případ (IFJ)

Jelikož jsou studentské projekty v předmětu Základy programování (IZP) přeci jen nižší komplexity rozhodl jsem se ve třetí fázi testování sáhnout po projektu z předmětu Formální jazyky a překladače (IFJ) pro školní rok 2019/2020. Tento projekt spočíval ve vytvoření překladače jednoduchého imperativního jazyka s formátováním ve stylu jazyka Python. Jako kód k testování jsem využil řešení týmu, jehož jsem byl členem, po souhlasu všech členů týmu.

Testovací sada čítala 14 zdrojových souborů. Knihovní soubory .h byly ze sady vynechány, z důvodů zaměření kontrol na ryze implementační kód. Jako vstupní konfigurace byla opět využita výchozí globální konfigurace (viz příloha B).

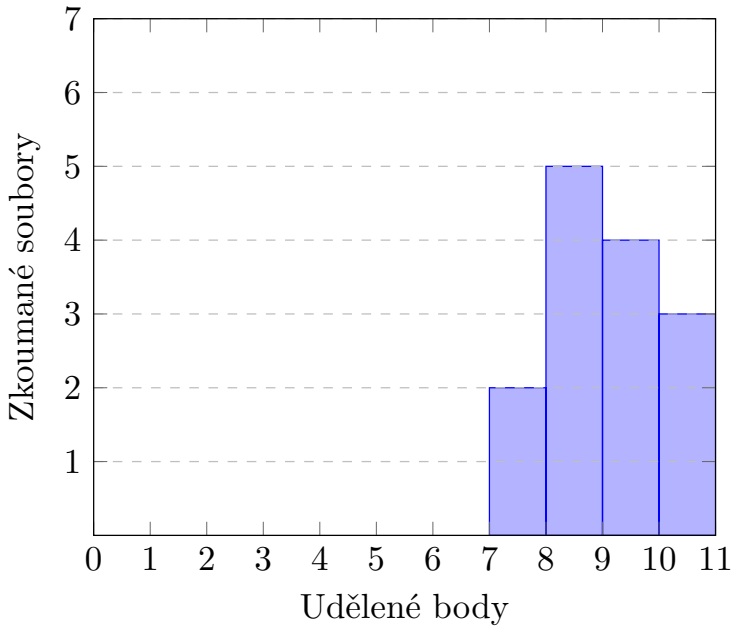
Tabulka 5.4: Výsledky testování IFJ projektu (14 souborů)

Kontrola	Počet výskytů	Průměrné množství
Nehodnocené kontroly (doporučení):		
readability-else-after-return	7x	3
readability-misleading-indentation	0x	0
readability-redundant-control-flow	0x	0
Nízká úroveň		
misc-redundant-expression	0x	0
misc-unused-parameters	0x	0
readability-function-size	2x	0.2143
readability-redundant-declaration	0x	0
readability-redundant-function-ptr-dereference	0x	0
readability-redundant-preprocessor	0x	0
Střední úroveň		
bugprone-redundant-branch-condition	0x	0
fit-c-macro-usage	3x	0.5714
fit-too-big-local-vars	0x	0
readability-braces-around-statements	11x	18.2857
readability-magic-numbers	5x	1.42857
Vysoká úroveň		
bugprone-infinite-loop	0x	0
fit-c-avoid-goto	0x	0

Výsledky testování potvrzují těžiště kolem využívání magických čísel a závorkování těl podmínek a cyklů. Obzvláště závorkování tvořilo u vybraných souborů vyšší desítky detekcí. Nejvíce detekcí obsahoval soubor `generator.c` (sémantická analýza + generátor instrukcí) s 83 výskyty, za ním následoval soubor `scanner.c` (stavový automat realizující lexikální analýzu) se 67 a třetí nejvyšší počet se vyskytoval v souboru `parser.c` (syntaktická analýza) s 53 výskyty. Nutno však podotknout, že v kontextu překladače je snaha redukovat množství kódu vynecháním závorkováním těl vcelku očekávatelnou.

Dva výskyty kontroly `readability-function-size`, způsobuje poprvé funkce pro inicializaci binárního stromu, která přesahuje limit parametrů (5 povolených vs 8 v kódu) a podruhé parser překonáním limitu pro větvení (20 povolených proti 24 v kódu).

Detekované využití maker je v tomto případě jednak využití konstant, tak i parametrických maker.



Obrázek 5.3: Histogram uděleného bodového hodnocení (IFJ)

Přesto však bodové hodnocení souborů zůstalo nadprůměrné a žádného souboru nekleslo pod hodnotu 7 bodů. Tuto skutečnost si vysvětluji několika faktory.

1. Větší zkušenností autorů. V době prací na projektu již měli autoři zkušenosti ze dvou úspěšně absolvovaných semestrů na VUT FIT a tím pádem i více zažité programátorské konvence.

2. Aktuálně dostupné kontroly jsou z velké části zaměřeny na začátečnické chyby a redundanci kódu. Nepodchycují již komplexnější problémy jako např. duplicitní kód, zřetězené volání funkcí, cyklomatickou složitost apod.

3. Vliv rozsahu kódu na finální hodnocení (viz 3.1.1).

Kapitola 6

Závěr

Cílem práce bylo vytvořit nástroj pro hodnocení dodržování zvolených programovacích konvencí a úhlednosti zdrojových kódů v jazyce C. Tento cíl se podařilo splnit.

Nástroj hodnotí předložené zdrojové kódy na základě dodané konfigurace, kde je možno specifikovat jaké kontroly se budou provádět, parametrizovat tyto kontroly (např. nastavování prahů) a přiřazovat jim váhu při finálním hodnocení. Analýza zdrojových kódů s využitím metod statické analýzy se provádí za pomoci nástroje Clang-Tidy, kterého program realizovaný touto prací využívá. Nástroj byl otestován jak na sadě umělých příkladů, tak i na reálných studentských projektech. Prvotní výsledky jsou uspokojivé, kdy je možno již nyní detekovat řadu začátečnických chyb způsobujících neúhlednost. Nestihl jsem však zpracovat některé kontroly jako například zřetězené volání funkcí, cyklomatickou složitost funkce a jiné.

Prostor pro budoucí rozšíření programu je značný. Jako první se jednoznačně nabízí možnost rozšiřování dostupné sady kontrol a to buď úpravami dosavadních Clang-Tidy kontrol i pro využití v C99 nebo tvorbou zcela nových. Dále pak rozšíření za účelem zvýšení uživatelského komfortu jako například přidáním grafického uživatelského rozhraní případně webového rozhraní pro ovládání programu. Přidání možnosti vizualizace výstupních dat. V neposlední řadě se také nabízí možnost začlenění programu do většího celku. To může být buďto začleněním do komplexnějšího řešení pro hodnocení úhlednosti, kde by byl program pouze částí realizující statickou analýzu. Nebo je možné jít i cestou integrace do některého populárního vývojového prostředí formou zásuvného modulu.

Literatura

[1] Al Ekram, R. a Kontogiannis, K. Source code modularization using lattice of concept slices. In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. Duben 2004, sv. 8, s. 195– 203. ISBN 0-7695-2107-X. Dostupné z: https://www.researchgate.net/publication/4065402_Source_code_modularization_using_lattice_of_concept_slices.

[2] Cooper, K. a Torczon, L. *Engineering a compiler*. 1. vyd. San Francisco ; London: Morgan Kaufmann, 2004. ISBN 1-55860-698-X. Dostupné z: https://books.google.cz/books?id=4yVQFVvsBNAC.

[3] Cooper, K. D., Kennedy, K. a Torczon, L. Compilers. In: Meyers, R. A., ed. *Encyclopedia of Physical Science and Technology (Third Edition)*. Third Edition. New York: Academic Press, 2003, s. 433–442. ISBN 978-0-12-227410-7. Dostupné z: https://www.sciencedirect.com/science/article/pii/B0122274105001265.

[4] Fang, X. Using a coding standard to improve program quality. In: SRA-KTL. *Proceedings Second Asia-Pacific Conference on Quality Software*. IEEE, 2001, s. 73–78. ISBN 0769512879.

[5] JetBrains. Code analysis on the fly. *CLion: A Cross-Platform IDE for C and C++ by JetBrains*. Leden 2021. Dostupné z: https://www.jetbrains.com/clion/.

[6] Kirkpatrick, R. *Tutorial* [online]. Logilab a PyCQA, únor 2021 [cit. 2021-02-22]. Dostupné z: http://pylint.pycqa.org/en/latest/tutorial.html.

[7] Microsoft. Vývoj. *Visual Studio 2019 IDE – software pro programování ve Windows*. 2019. Dostupné z: https://visualstudio.microsoft.com/cs/vs/. Path: Visual Studio; Další informace.

[8] Møller, A. a Schwartzbach, M. I. *Static Program Analysis*. November 2020. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

[9] PyCQA. *Pylint User Manual* [online]. Uživatelský manuál, 2.8.0. PyCQA, únor 2021 [cit. 2021-03-23]. Dostupné z: http://pylint.pycqa.org/en/latest/index.html.

[10] Smrčka, A. *ITS - Testování se znalostí zdrojových kódů, datové toky*. Brno: [b.n.], 2020 [cit. 2021-03-23]. Přednáška, snímek 4,14. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FITS-IT%2Flectures%2F3-dataflow.pdf&cid=13359.

[11] SonarSource. Developer Edition. *Download|SonarQube*. Květen 2021. Dostupné z: https://www.sonarqube.org/downloads/. Path: Developer Edition.

[12] Team, T. C. *Extra Clang Tools 13 documentation*. 13. vyd. 2021 [cit. 2021-05-01]. Dostupné z: https://clang.llvm.org/extra/clang-tidy/.

[13] Vojnar, T. *Statická analýza a verifikace - Úvod, základní pojmy*. Brno: [b.n.], 2020 [cit. 2021-04-30]. Přednáška, snímek 22. Dostupné z: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-01.pdf.

[14] YAML. 3.2.1.1. Nodes. *YAML Ain't Markup Language (YAML™) Version 1.2*. Říjen 2009. Https://yaml.org/spec/1.2/spec.html#id2764044. Dostupné z: https://yaml.org/spec/1.2/spec.html.

[15] YAML. Projects:. *The Official YAML Web Site*. 2018. Dostupné z: https://yaml.org/.

Výpisy

Seznam obrázků

Seznam tabulek

Příloha A

Použité Clang-Tidy kontroly

Tato přílohy slouží k popsání jednotlivých Clang-Tidy kontrol, jenž aplikace využívá. Popisy jednotlivých kontrol vycházejí z textů oficiální dokumentace .

A.1 Vestavěné kontroly

bugprone-infinite-loop hledá zjevné nekonečné cykly (tj. takové kde se řídící proměnná cyklu nemění)

bugprone-redundant-branch-condition hledá proměnné v zanořených podmínkách, které však již byly testovány ve vnější podmínce a jejich stav se nezměnil

misc-unused-parameters hledá nevyužité parametry funkce

misc-redundant-expression detekuje redundantní výrazy, typicky způsobené chybou při kopírování.Detekuje výrazy které jsou redundantní, vždy pravdivé, vždy nepravdivé a vždy 0 nebo 1.

readability-braces-around-statements kontroluje ozávorkování těl podmínek a cyklů.

readability-function-size tato kontrola sleduje vícero aspektů funkce a to konkrétně:

LineThreshold maximální povolený počet řádků kódu,

StatementThreshold maximální povolený počet příkazů,

BranchThreshold maximální povolený počet větví,

ParameterThreshold maximální povolený počet parametrů,

NestingThreshold maximální povolený počet zanoření,

VariableThreshold maximální povolený počet proměnných.

readability-magic-numbers detekuje magická čísla, tj. celočíselné nebo desetinné literály, jenž jsou natvrdo zapsány v kódu a nepochází z konstant nebo symbolů.

readability-redundant-declaration detekuje redundantní deklarace proměnných a funkcí.

readability-redundant-function-ptr-dereference detekuje redundantní dereference ukazatelů.

readability-redundant-preprocessor detekuje redundantní preprocesorové direktivy.

A.2 Upravené kontroly

fit-avoid-goto detekuje užívaní nepodmíněných skoků. Jediné povolené užití je skok dopředu uvnitř smyčky.

fit-macro-usage detekuje používání maker pro definci konstant, parametrická makra a variadická makra.

A.3 Vlastní kontroly

fit-too-big-local-vars hledá lokální proměnné přesahující svou velikostí stanovenou maximální mez.

A.4 Nehodnocené kontroly (doporučení)

readability-else-after-return prosazuje část LLVM kódovacího standardu o omezení `else`, `else if` větví pokud předchozí pravdivá větev obsahuje `return`, `brake` nebo `continue`.

readability-misleading-indentation detekuje podezřelá odsazení, která mohou být způsobena chybějícím ozávorkováním.

readability-redundant-control-flow detekuje redundantní výrazy jako např. užití `return` v beznávratové funkci.

Příloha B

Konfigurace využitá pro testování

Zdejší konfigurace použitá při testování se nachází v souboru `revisor_default_config.yaml`

```
static_analysis:
  clang-tidy:
    checks:
      # basic set of checks (13+5-1)
      bugprone-infinite-loop:
        weight: 3
      bugprone-redundant-branch-condition:
        weight: 2
      fit-c-avoid-goto:
        weight: 3
      fit-c-macro-usage:
        weight: 2
      fit-too-big-local-vars:
        weight: 2
      readability-braces-around-statements:
        weight: 2
      misc-unused-parameters:
        weight: 1
      misc-redundant-expression:
        weight: 1
      readability-function-size:
        param:
          LineThreshold: 200
          BranchThreshold: 20
          ParameterThreshold: 5
          NestingThreshold: 10
          VariableThreshold: 30
        weight: 1
      readability-magic-numbers:
        weight: 2
      readability-redundant-declaration:
        weight: 1
```

```
      readability-redundant-preprocessor:
        weight: 1
      readability-redundant-function-ptr-dereference:
        weight: 1
      # recommendation checks (not included in evaluation; should have
          weight 0)
      readability-misleading-indentation:
        weight: 0
      readability-redundant-control-flow:
        weight: 0
      readability-else-after-return:
        weight: 0
      # functional checks (optional)
# bugprone-suspicious-include:
# weight: 0
# bugprone-suspicious-semicolon:
# weight: 0
# weight: 0
```

Příloha C

Testování - příklad bubble_sort.c

C.1 Umělý příklad bubble_sort.c

```
#include <stdio.h>
extern int j;
extern int j; // redundantni deklarace
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n-1; j++) {// preklep v inkrementaci => nekonecny
       cyklus
       for (j = 0; j < n-i-1; j++){
          if (arr[j] > arr[j+1]) {
            int tmp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = tmp;
          }
       }
   }
}
int main(main(int argc, char *argv[])) { // nevyuzity parametr argc
    int cond = argv[1];
    int cond2 = argv[2];
    int arr[] = {18, 38, 33, 89, 68, 48, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (cond) {
      if (cond) { // redundantni podminka, jiz byla kontrolovana na
          prechozi urovni
        bubbleSort(arr, n);
      }
    }
    int a = 0;
    output: // cyklus realizovany goto
    if (a < n) {
      printf("%d ", arr[a]);
    }
    else // chybejici ozavrkovani tela else vetve
```

```
      goto output;
    return 0;
}
```

C.2 Konfigurace pro `bubble_sort.c`

```
static_analysis:
  clang-tidy:
    checks:
      bugprone-infinite-loop:
        weight: 3
      bugprone-redundant-branch-condition:
        weight: 2
      fit-c-avoid-goto:
        weight: 3
      readability-braces-around-statements:
        weight: 2
      misc-unused-parameters:
        weight: 1
      readability-redundant-declaration:
        weight: 1
```

C.3 Výstup revisor příkladu bubble_sort.c

```
Examining ../debug_files/bubble_sort.c :
/home/user/bubble_sort.c:3:12: warning: redundant 'j' declaration [
    readability-redundant-declaration]
extern int j;
~~~~~~~~~~~^~
/home/user/bubble_sort.c:2:12: note: previously declared here
extern int j;
           ^
/home/user/bubble_sort.c:5:4: warning: this loop is infinite; none of its
    condition variables (i, n) are updated in the loop body [bugprone-
    infinite-loop]
   for (int i = 0; i < n-1; j++) {
   ^
/home/user/bubble_sort.c:15:5: error: first parameter of 'main' (argument
    count) must be of type 'int' [clang-diagnostic-error]
int main(main(int argc, char *argv[])) {
    ^
/home/user/bubble_sort.c:15:10: warning: parameter 'main' is unused [misc-
    unused-parameters]
int main(main(int argc, char *argv[])) {
         ^
/home/user/bubble_sort.c:16:16: error: use of undeclared identifier 'argv'
    [clang-diagnostic-error]
    int cond = argv[1];
```

```
              ^
/home/user/bubble_sort.c:17:17: error: use of undeclared identifier 'argv'
    [clang-diagnostic-error]
    int cond2 = argv[2];
                ^
/home/user/bubble_sort.c:21:7: warning: redundant condition 'cond' [
    bugprone-redundant-branch-condition]
      if (cond) {
      ^~~~~~~~~~
/home/user/bubble_sort.c:30:9: warning: statement should be inside braces [
    readability-braces-around-statements]
    else
        ^
         {
/home/user/bubble_sort.c:31:7: warning: avoid using 'goto' for flow control
     [fit-c-avoid-goto]
      goto output;
      ^~~~~~~~~~
/home/user/bubble_sort.c:26:5: note: label defined here
    output:
    ^


--------------------------------------------------------------------------------

Checks result:
................................................................................

Recommendations (not evaluated):

................................................................................

Low severity:

misc-unused-parameters : FAILED (1x)
readability-redundant-declaration : FAILED (1x)
................................................................................

Medium severity:

bugprone-redundant-branch-condition : FAILED (1x)
readability-braces-around-statements : FAILED (1x)
................................................................................

High severity:

bugprone-infinite-loop : FAILED (1x)
fit-c-avoid-goto : FAILED (1x)
```

```
------------------------------------------------------------------------------

Rating:
6.36364 points out of 10

******************************************************************************

******************************************************************************
```